Robust Service Function Chains in OpenStack

Giuseppe Di Lena

▶ **To cite this version:**

Giuseppe Di Lena. Robust Service Function Chains in OpenStack. Networking and Internet Architecture [cs.NI]. 2017. hal-01651440

HAL Id: hal-01651440
https://inria.hal.science/hal-01651440

Submitted on 29 Nov 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Université Côte D'Azur

Master II IFI

International track UBINET

Robust Service Function Chains in OpenStack

Author:
Giuseppe Di Lena

Supervisors:
Damien Saucez
Thierry Turletti

August 31, 2017

Contents

Abstract

Cloud Computing and virtualization are largely adopted in the IT world and starts to gain momentum in networking to implement network functions such as switches or routers but also complex network service chains such as end to end security services. Surprisingly, while robustness is the first question that comes when deploying dedicated network infrastructures, cloud infrastructures provide little or no robustness guarantees for virtualized network functions.

As a matter of facts, traditional Cloud infrastructure are several order of magnitude slower than dedicated network infrastructures to react to faults. In this work, we aim at providing robustness guarantees for network functions deployed with OpenStack, the leading open source project for cloud computing.

Multiple options are possible to implement such robustness in OpenStack but we designed an approach that minimizes the changes required in OpenStack in order such that it can be smoothly integrated with the official releases. Our well documented implementation follows the best programming practices to guarantee its correctness and integration in a large open source project. Our tests in a real OpenStack environment composed of multiple compute nodes validate our choices and prove its usability.

1 Introduction

1.1 Context

Cloud computing is a set of technologies and infrastructures that allow convenient and on-demand access to shared computing resources (servers, networks, storage, applications, and services) without the end user knows the physical location, the infrastructure, and configuration of the system providing the service[1].

Typically, the service offered to the public is pay-for-use. The Cloud Provider has software resources, depending on the service it wants to deliver, and hardware resources (servers and clusters) located in various data centers spread across different parts in the world, connected to each other through private high-speed networks.

Three common cloud delivery models have become widely established and formalized; each model should satisfy the requirements of a particular class of users:

1. Software as a Service (SaaS): the provider enables the users to use applications that are executed on its servers; The user can access the application using a client like a PC, smartphone, tablet, etc.

   In this case, the user does not have to worry about the configuration or the installation of the application; the Cloud provider has to maintain and update the applications executed on the servers, the operating system, and the computational resources.

2. Platform as a Service (Paas): the provider offers a platform where the users can develop and deploy their applications. The users do not have to worries about system configuration at the lower level like Operating system, storage or hosting system;

3. Infrastructure as a Service (IaaS): the provider offers a virtual infrastructure (CPU, memory, storage, and network) that the users can use in place of or with the physical infrastructure. The users have to configure the operating system, the environment, and the network.

During this project, we work with OpenStack, an open source project for cloud computing, typically for IaaS clouds.[2]

1.2 Motivations

In the present days, companies start to move Network functions to the Cloud; Networks are built in reality to be robust at failures (if one network device between 2 hosts filed, the hosts must be reachable each other).

Up to now, cloud infrastructures provide elasticity, but not robustness.

1.3 Goal

The goal of this project is not to design and propose new algorithms to guarantee the robustness of a chain(it is not a research project). The goal is to build a stable OpenStack environment where it is possible to implement robustness solution that have been proposed in the literature without too much effort.

1.4 Challenges

OpenStack is a big and complex project, up to now it has around 4 million Line Of Code, more than six thousand contributors, and more than 400 thousand commits.[3]

One of the challenges is that a new version of OpenStack is released every six months with significant changes from the previous version. We want that in case the Community decides to modify OpenStack, the algorithm should be reimplemented with the minimum effort.

The documentation for the developer is limited(it is hard to find a true class or sequence diagram), and usually, it is not updated.

1.5 Contributions

Produce a documentation for new developers to contribute in OpenStack (especially in the Nova Project). Propose different solutions in order to add Robustness in the OpenStack infrastructure, without compromise the main functionalities; finally, implement one solution and publish all the code and the documentation on a public repository [15] that allow other developers to contribute.

2 State Of The Art

2.1 OpenStack

OpenStack is an open source project for cloud computing; it is composed of seven different main modules. Each module provides APIs so administrators can manage resources through dashboards, while users can access them through a web interface, a command-line client, or a software development kit that supports APIs.[4]

OpenStack is mainly written in Python with a modular architecture, it is released under the terms of the free Apache License and is currently maintained and supported by the OpenStack Foundation, a non-profit entity founded in September 2012.

The OpenStack community collaborates around a six-month, time-based release cycle with frequent development milestones. During the planning phase of each release, the community gathers for an OpenStack Design Summit to facilitate developer working sessions and to assemble plans.

Hundreds of the world's largest companies rely on OpenStack to run their businesses every day, reducing costs and helping them move faster. OpenStack has a strong ecosystem, and users seeking commercial support can choose from different OpenStack-powered products and services in the Marketplace.

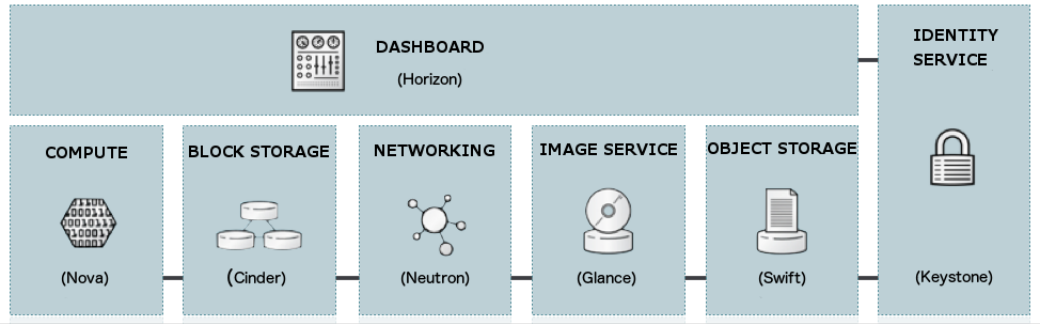

Figure 1: OpenStack Modules [4]

2.1.1 Nova

Nova (OpenStack compute) is a cloud computing fabric controller, which is the main part of an IaaS system. It is designed to manage and automate pools of computer resources and can work with widely available virtualization technologies.

Probably Nova is the most famous among the OpenStack projects, the most complicated and the most distributed. It provides virtual servers on request. A large number of processes work together to transform end-user API requests into running virtual machines.[5]

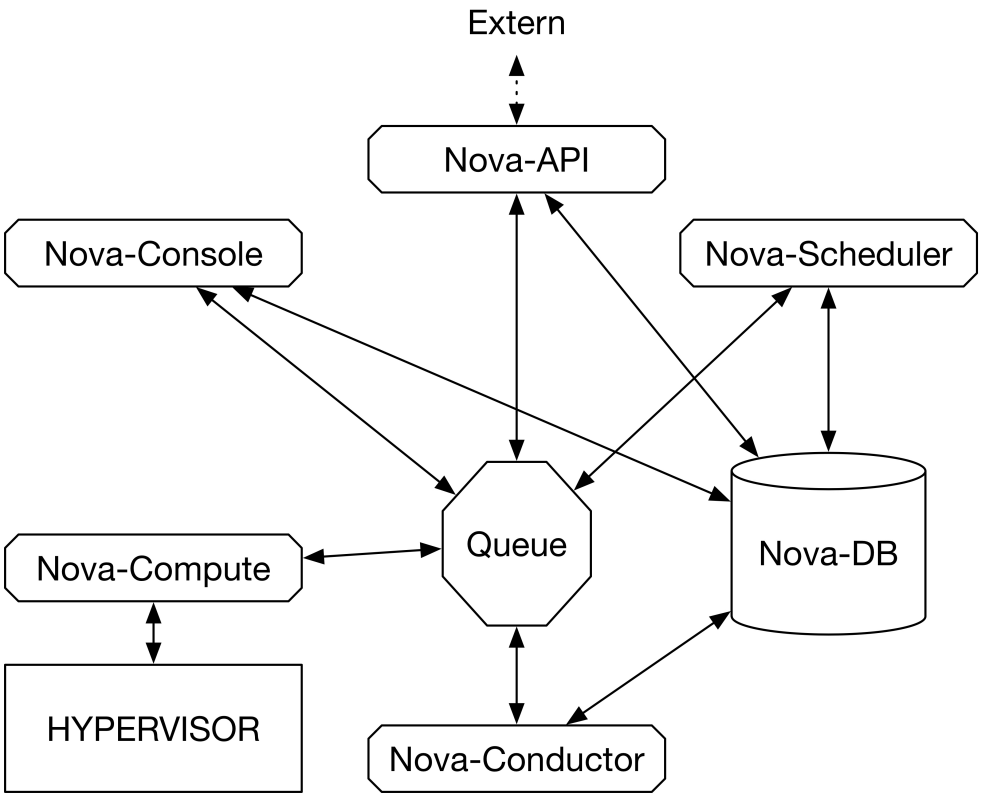

Figure 2: Nova Architecture

Nova is composed of seven different modules:

1. Nova-API: a REST-ful API that accepts incoming commands and interacts with the OpenStack Cloud;
2. Nova-compute: a daemon that creates and destroys virtual machine instances via the HYPERVISOR API;
3. Nova-scheduler: takes a request from the queue and chooses where the virtual instance should be executed;
4. Nova-conductor: provides nova-compute services(for example create/delete instances) interacting with the Database;
5. Nova-DB: collects most of the build-time and run-time information;
6. Nova-console: it provides console services to allow the end user or the administrator to access the console of its virtual instances through a proxy.
7. The queue works as a central hub for exchanging messages between daemons. Normally this is implemented with RabbitMQ;

It is important to know that we can receive user commands from Horizon (Web Interface) and Nova-Client(an independent project, that provides to the users a Command Line Interface to interact with Nova-API).

2.1.2 Neutron

OpenStack Networking is a pluggable, scalable, and API driven system to manage networks and IP addresses in an OpenStack-based cloud. Like other core OpenStack components, OpenStack Networking can be used by administrators and users to increase the value and maximize the utilization of existing data center resources.[6]

Neutron, the code name of OpenStack Networking, is a standalone service that can be installed independently of other OpenStack services. Neutron includes many technologies one would find in a data center, including switching, routing, load balancing, firewalling, and virtual private networks.
These features can be configured to leverage open-source or commercial software;[7] Neutron also provides a framework for third-party vendors to build on and enhance the capabilities of the cloud.

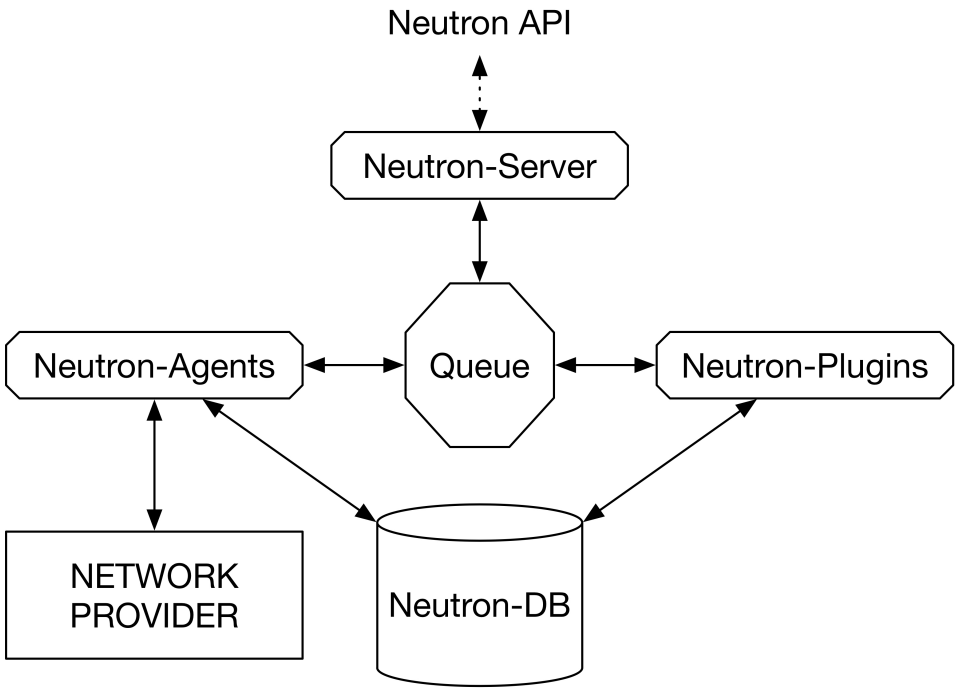

Figure 3: Neutron Architecture

Neutron is composed of four different modules[8]:

1. Neutron-Server: this service runs on the network node to service the Networking API and its extensions. It also enforces the network model and IP addressing of each port.

2. Neutron-Agent: runs on each compute node to manage local virtual switch (vswitch) configuration. The plug-in that one use determines which agents run. This service requires message queue access and depends on the plugin used.

3. Neutron-Plugin: the neutron-server requires indirect access to a persistent database. This is accomplished through plugins, which communicate with the database using AMQP.

4. Neutron-DB: keep track of the networks created

2.1.3 Keystone

Keystone is the Identity Service module; it manages the users and all the privileges that they have on the entire infrastructure and the services that are implemented in OpenStack[9]. Keystone provides a central directory where users are mapped to the services they can access. It supports several authentication methods: user and password authentication pair, token-based authentication systems, certificate-based authentication (like that of Amazon EC2, based on X.509 certificates).

The key concepts in Keystone are:

- User: Digital representation of a person, system or service using OpenStack; Each user has credentials (user name and password, username and API key, or a Keystone token).
- Token: An alphanumeric string that is used in order to access a particular resource; each token has a scope that describes what resources are accessible with it. It has a limited duration and can be revoked at any time.(it is very similar to the Kerberos protocol)
- Tenant: A user group that has shared resources.
- Endpoint: An accessible network address, usually described by a URL, in order to use a service.
- Service: an OpenStack service (Nova, Swift, Neutron, etc.) that provides one or more endpoints which users can access resources and perform operations.
- Role: rights and privileges that a user gains, allowing them to perform a specific set of operations and requesting certain resources. One user can assume different roles in different tenants and assume multiple roles within the same tenant.

If a user wants to start an OpenStack service (for example, creating a virtual server) first, they will have to authenticate using a keystone user name and password. If the credentials are correct, Keystone will send the user a token that will allow them to use the services it has access to. A token is always temporary, if it expires and the user requires another service, he must first renew it.

2.1.4 Glance

Glance provides discovery, registration, and delivery services for disk and server images. Stored images can be used as a template. It can also be used to store and catalog an unlimited number of backups[2]. Glance API provides a standard REST interface for querying information about disk images and lets clients stream the images to new servers.

2.1.5 Swift

Swift is a scalable, redundant storage system. Objects and files are written to multiple disk drives spread throughout servers in the data center, with the OpenStack software responsible for ensuring data replication and integrity across the cluster. Storage clusters scale horizontally simply by adding new servers. Should a server or hard drive fail, OpenStack replicates its content from other active nodes to new locations in the cluster[2]. Because OpenStack uses software logic to ensure data replication and distribution across different devices, inexpensive commodity hard drives and servers can be used.

2.1.6 Cinder

Cinder provides persistent block-level storage devices for use with OpenStack compute instances. The block storage system manages the creation, attaching and detaching of the block devices to servers. Block storage volumes are fully integrated into OpenStack Compute and the Dashboard allowing for cloud users to manage their own storage needs. Block storage is appropriate for performance sensitive scenarios such as database storage, expandable file systems, or providing a server with access to raw block level storage[2]. Snapshot management provides powerful functionality for backing up data stored on block storage volumes. Snapshots can be restored or used to create a new block storage volume.

2.1.7 Horizon

Horizon provides a web interface developed using Django, an open-source framework written in Python, to interact with OpenStack services depending on the privileges you have, there are two types of interfaces:

- User interface: allows for each project to which the user is assigned: to launch virtual machine instances, create virtual networks, and apply

firewall rules, to have information about the resources available for each project, to create images and snapshots of virtual machines and simply switch from one project to another.

- Administrator Interface: In addition to user interface operations, we can set up and manage all projects, create and manage users (for example, assign a user to a project). We can get all system information (such as services Active in the Cloud and quotas allocated to each project), create new flavors and manage system resources.

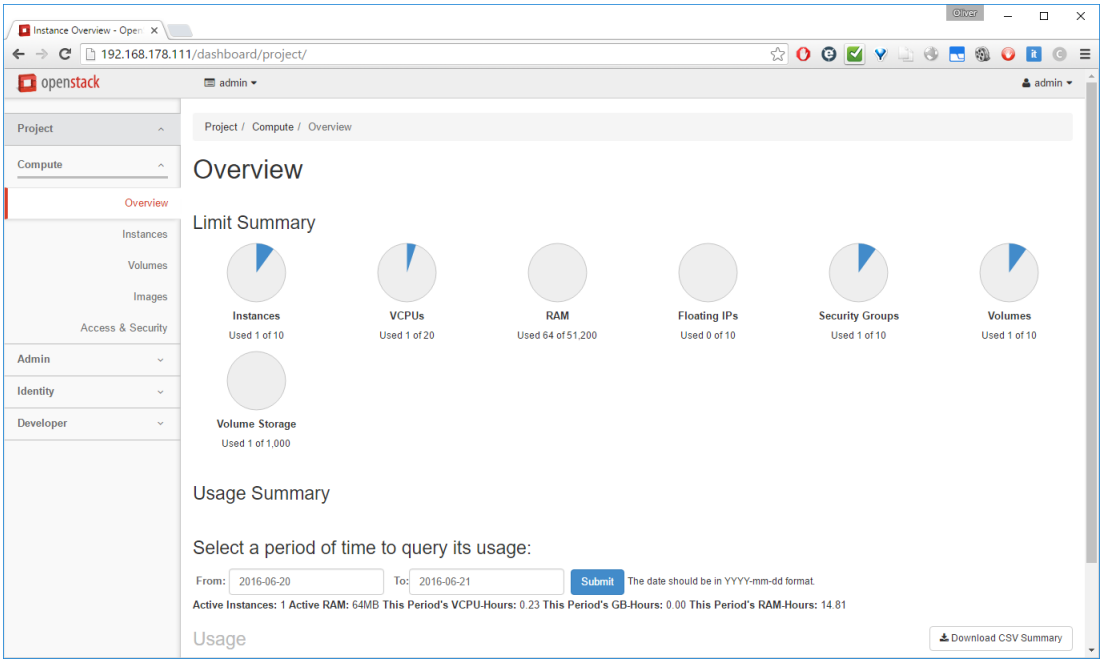

Figure 4: Dashboard

2.1.8 Heat

Heat is the main project in the OpenStack Orchestration program. It implements an orchestration engine to launch multiple composite cloud applications based on templates in the form of text files that can be treated like code[10].

Heat is not a primary OpenStack project like the ones described before, but it can be easily integrated with the OpenStack environment.

The key concepts in Heat are:

- Resources: Objects that will be created or modified during the orchestration. Resources can be networks, routers, subnets, instances, volumes, floating IPs, security groups and more.
- Stack: In Heat, a stack is a collection of resources, for example a Network function chain.
- Parameters: Allow the user to provide input to the template during deployment. For example, if you want to input the name for an instance in an orchestration, that name could be input as a parameter in the template and change during each run time.
- Templates: How a Stack is defined and described with code.

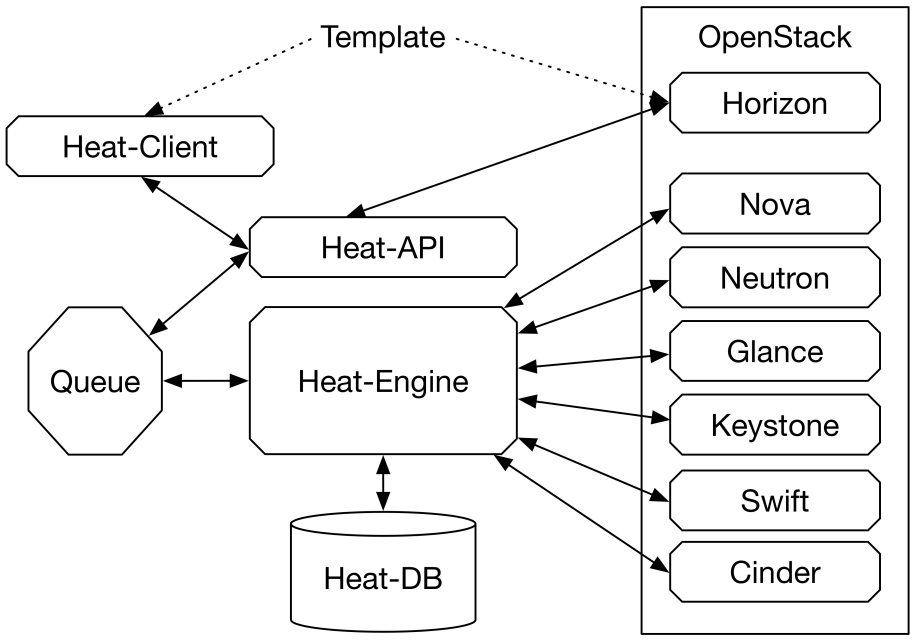

Figure 5: Heat Architecture

There are four main components of the Heat project:

1. Heat-Client: is the CLI that communicates with the Heat-API(Heat is also integrated with Horizon, so it is possible to communicate with Heat-API, through the Web interface).

2. Heat-API: The component that provides an OpenStack-native REST API that processes the requests and sends them to the heat-engine.

3. Heat-Engine: is the brains of the operation and does the main work of orchestrating the launch of templates and providing events back to the API consumer.

4. Heat-DB: Store all the information about the Stacks created.

2.2 NFV & SNFC

Network Functions Virtualization(NFV) is a concept that uses the technologies of IT virtualization to virtualize entire classes of network node functions into building blocks that may connect, or chain together, to create communication services[11].

NFV describes and defines how network services are designed, constructed and deployed using virtualized software components and how are decoupled from the hardware upon which they execute.

Network functions virtualization directly addresses most of the limitations associated with using the traditional network and brings many additional benefits. It offers a framework to completely transform the way networks are architected, deployed, managed, and operated while offering many layers of improvement and efficiency across all of these[12].

Why use NFV:

- Hardware Flexibility: NFV uses regular hardware, network operators have the freedom to choose and build the hardware in the most efficient way to suit their needs and requirements.

- Scalability and Elasticity: For scaling up the traditional network equipment's capacity takes time, planning, and money. This problem is solved by NFV, which allows capability changes by offering a means to expand and shrink the resources used in the Cloud. If any of the Virtual Network Function(VNF) requires additional CPU, storage, or

bandwidth, it can be requested and allocated to the VNF from the hardware pool. In a traditional network device, it would require either a full device replacement or a hardware upgrade to alter any of these parameters.

- Rapid Development: NFV provides the means to easily deploy a different vendor's solution without the heavy costs associated with replacing an existing vendor's deployment, it keeps network operators from being locked into a particular vendor.

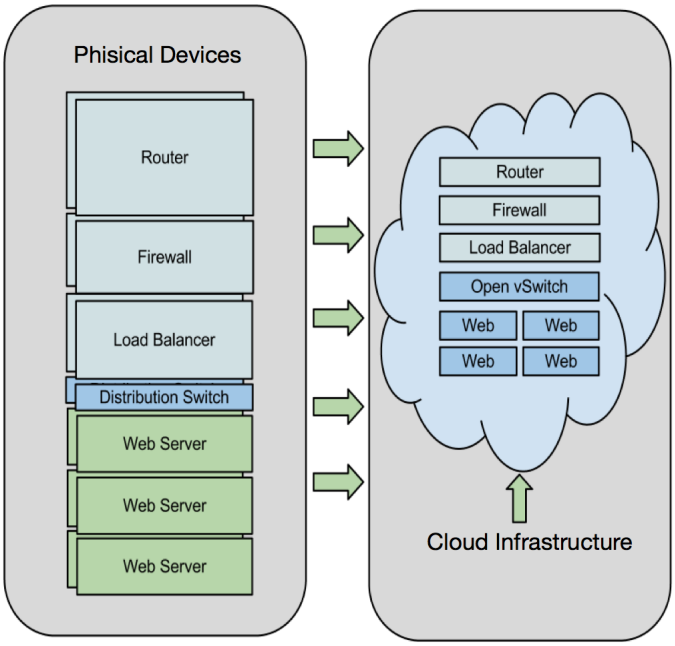

Figure 6: Network Function Virtualization

Service Network Function Chain (or Service Function Chaining) provides the ability to define an ordered list of network services (e.g., firewalls, load balancers). These services are then grouped together in the network to create a service chain. The "chain" in service chaining represents the services that can be connected across the network using software provisioning. Network service chaining capabilities mean that a large number of virtual network functions can be connected together in an NFV environment.

Because it's done in software using virtual circuits, these connections can be set up and torn down as needed with service chain provisioning through the NFV orchestration layer.

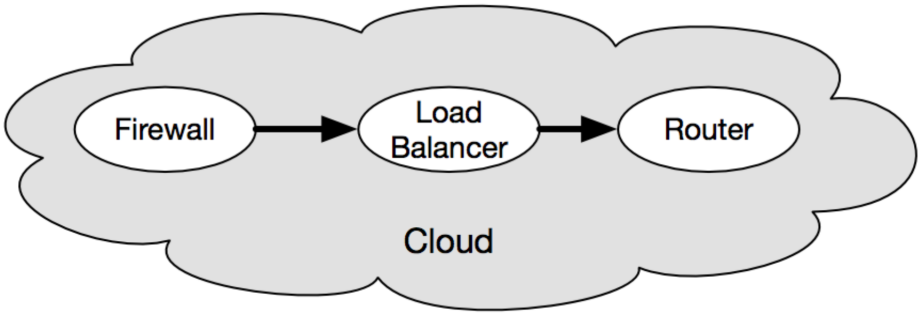

Figure 7: Service Network Function Chain

2.3 Robustness

In this section, we define the concept of Robustness related to Service Network Function Chain. Each Instance(Virtual Function) of the chain must run in a physical Host(Compute Node) inside the Cloud infrastructure; If the Host that runs the Instance fails(for example a hardware failure), the chain is not working anymore.

- Robustness: an Integer number bigger or equal than 0, assigned to an SNFC indicating the minimum number of simultaneous Compute failure inside the Cloud infrastructure such that the Chain is not compromised and it is serving the users without breaks.

For example, if we instantiate a chain with robustness 2 and two compute nodes in the Cloud infrastructure crash, for the robustness property, we are sure that the chain is working. If we have three crashes, we do not have the guaranty that the chain is still working.

There is a lot of work around SNFC and robustness, as we said before, the goal of this work is not to provide new algorithm or new definitions of Robustness; For this reason, we are not going to explain in detail how to guaranty

Robustness in a system; we just present the main properties(Redundancy and placement) that a Cloud infrastructure needs in order to guaranty the robustness described in the paper *"To Split, or not to Split: When SFC Robustness is at Stake"*[13].

The idea is to add redundancy; let's take in consideration a chain with a given robustness = 1 (so it has to be still available if one Compute breaks in the system), we can think to create two chains to guarantee the robustness. Is it enough in order to guaranty the robustness?

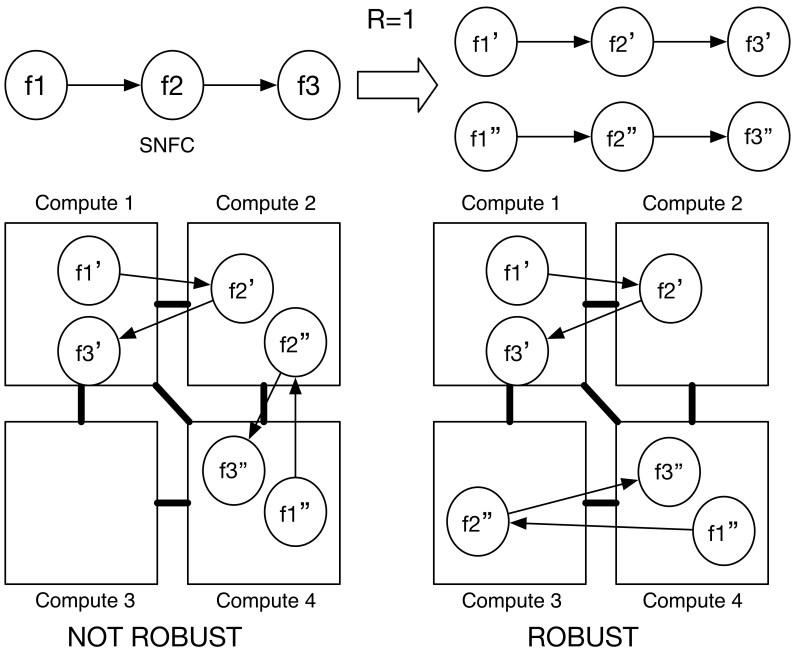

Figure 8: Allocation Chain Example

In this example, we can see that redundancy is not enough; if Compute 2 breaks in the first case, we can see that the first copy of the chain (f') and the second copy of the chain(f"), are not working anymore. In the second sample if Compute 2 breaks, the copy f' is not running anymore, while the copy f" is working, because the crash of Compute 2 does not have any effect for f".

So in order to guaranty Robustness, we need redundancy, and we need to place the functions of the chain correctly in the Compute Nodes.

3 Our Approach

3.1 Solution Proposed

OpenStack provides various API for developers; in this project, we focalized in the API provided by Horizon and Nova. When an instance is created, the message that Nova receives from the API contains multiple properties and different options(Network, vCPU, Memory, etc).

For our purposes, we focus on two properties in the function called when an instance is created[14]:

- AntiAffinityGroup Option: Anti-Affinity groups allow us to make sure instances are on different Hosts.
- Instance Hints: Hints allow us to send to Nova personalized information that are not provided normally(for example the robustness of a chain)

3.1.1 Solution 1

A first simple possible solution, it is just to modify Horizon, without affecting the core of OpenStack. When a User creates an instance(or a chain) with an optional parameter (Instance Hint "Robustness:Integer"), we intercept the call in Horizon, before that it is sent to Nova and we modify the call in this way:

- Create Robustness + 1 (or more)copy of the instance.
- Create a new Anti-Affinity group.
- Put the instances in the same Anti-Affinity group.

In this way, we are sure that the instances are in different Compute Host(Anti-Affinity Group), and we create all the copy that we need to ensure a robust instance. In the case of a Chain, this process is executed for any function of the Chain.

PRO:

- Simple to Implement.
- Correctness of the Anti-Affinity Group is proven, it is part of Nova release.

CONS:

- Impossible to implement directly complex robustness algorithms(we do not have total control of the Scheduler).
- We don't have control of the Anti-Affinity Group option; if the community decide to not support anymore this option, we have to reimplement the entire solution.
- Horizon is not the only way to create an instance(or a chain); we need to modify also Nova-Client(Command Line Client for Openstack)

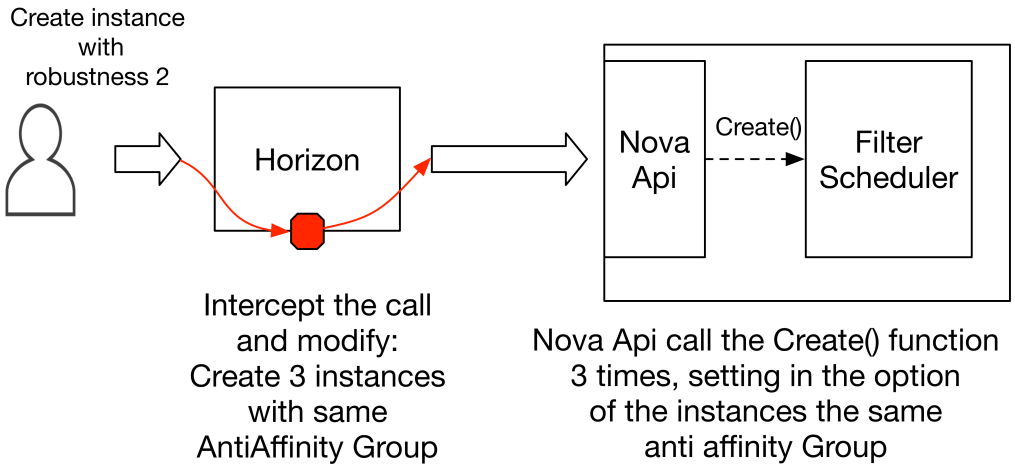

Figure 9: Changes in Horizon to allow robustness

3.1.2 Solution 2

In this case we need to go in the core of OpenStack; in particular, we modify the Nova-API, and we create a new scheduler without modifying or delete the default scheduler(FilterScheduler).

When Nova receives the message for the creation of an instance, first we check that in the properties of the instance there is the Hint "Robustness", if the property is not there, call proceed regularly(default scheduler and one instance created). If the Hint "Robustness" is specified and it is an

integer >= 1, the control goes to our Personalized Scheduler(that chooses the number of copies to create and where to place them).

PRO:

- We have total control of the Personalized Scheduler.
- If the community decide to modify the Default scheduler our solution will continue to work
- In this solution we modify only Nova.

CONS:

- Nova is a big and complex project, it is difficult to modify.
- Change the API
- Dependency on the implementation of Nova (while previous option just depends on the API)

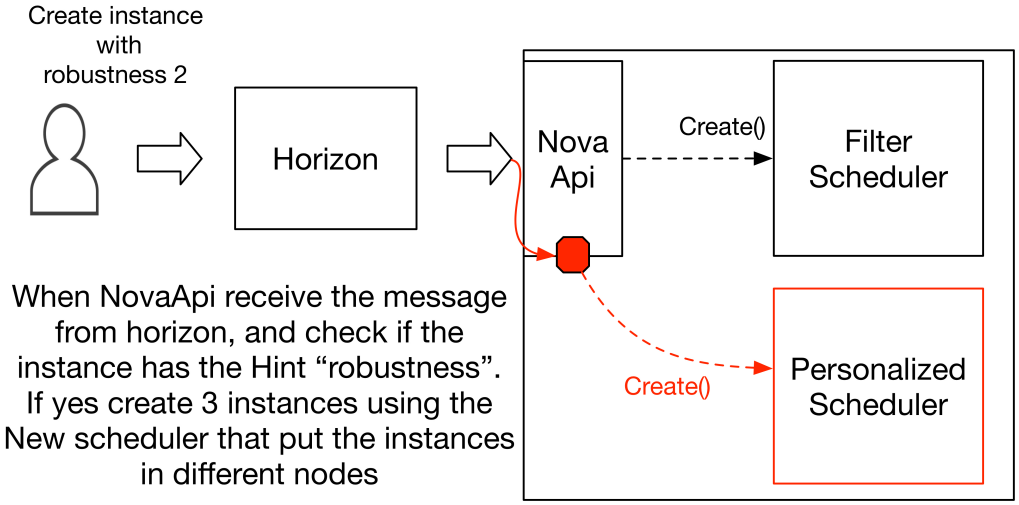

Figure 10: Changes in Nova to allow robustness

3.1.2.1 Possible Extension It is possible to extend this solution to enhance the integration with OpenStack, building a new API in Horizon and Nova. We build a new API in Horizon to support the Robustness option, and we create a new API in Nova to use our personalized scheduler.

PRO:

- We have total control of the Personalized Scheduler.
- No problem if the community decide to modify the Default Scheduler or Horizon

CONS:

- Horizon is not the only way to create an instance(or a chain); we need to create a new API also for Nova-Client(Command Line Client for OpenStack).
- we have to create one API for Nova, Horizon and for Nova-Client.

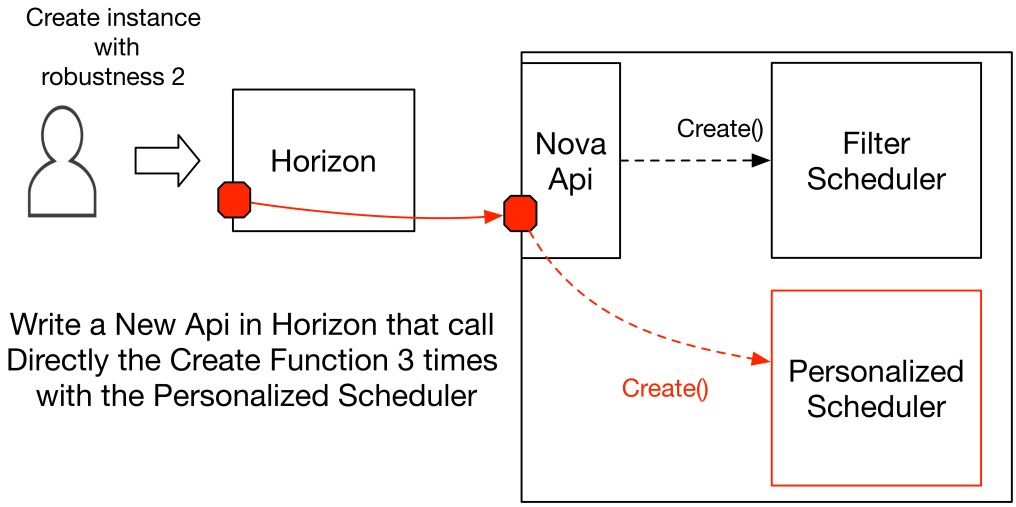

Figure 11: Integrate new API in Horizon and Nova

3.2 Implementation of Sol. 2

Knowing that OpenStack is evolving very fast, and we need a solution that is durable and adaptable to the new version of OpenStack easily. We decide to modify Nova (the core of Nova is stable, it is difficult that the community decides to change its architecture, most of the commits in the Nova project are for debugging); without affecting the Horizon API(it is common that the community decides to modify the API, for example, to support new features).

3.2.1 Allocation Scheduler

First of all, we create a new scheduler. Each scheduler is a python file, saved in the directory *nova/scheduler/SCHEDULER_NAME.py*.

A scheduler in OpenStack has a particular structure, defined in the file *nova/scheduler/driver.py*, each scheduler defined must inherit the class *Scheduler* in the *driver.py* file, in order to work correctly.

```
### driver.py

import abc
import six
from stevedore import driver
import nova.conf
from nova import objects
from nova import servicegroup

CONF = nova.conf.CONF

@six.add_metaclass(abc.ABCMeta)
class Scheduler(object):
    """The base class that all Scheduler classes should
                                inherit from."""

    def __init__(self):
        self.host_manager = driver.DriverManager(
                "nova.scheduler.host_manager",
                CONF.scheduler_host_manager,
                invoke_on_load=True).driver
        self.servicegroup_api = servicegroup.API()

    def run_periodic_tasks(self, context):
```

```
        """Manager calls this so drivers can perform periodic
                                    tasks."""
25          pass

27      def hosts_up(self, context, topic):
            """Return the list of hosts that have a running
                                    service for topic."""
29
            services = objects.ServiceList.get_by_topic(context,
                                    topic)
31          return [service.host
                    for service in services
33                  if self.servicegroup_api.service_is_up(
                                    service)]

35      @abc.abstractmethod
        def select_destinations(self, context, spec_obj):
37          """Must override select_destinations method."""
            return []
```

To test our infrastructure, the scheduler that we implemented is simple, and it does not take much effort to modify it to implement a different scheduler. If someone has to implement its own scheduler, it has just to modify the method **_schedule** (Line: 19). The *RRScheduler* returns the list of hosts that can satisfy the request of the Instance(for example if the compute has enough vCPUs or vRAM); after the filtering process (**_filter_hosts** method Line: 27), it sort the compute nodes by the IDs of the compute nodes and return the sorted list to the **select_destinations** method (Line: 34).

```
### Personalized Scheduler
2
import nova.conf
4 from nova import exception
from nova.i18n import _
6 from nova.scheduler import driver
CONF = nova.conf.CONF
8
class RRScheduler(driver.Scheduler):
10      """RR scheduler"""

12      def _filter_hosts(self, hosts, spec_obj):
            """Filter a list of hosts based on RequestSpec."""
```

```
14
        ignore_hosts = spec_obj.ignore_hosts or []
16      hosts = [host for host in hosts if host not in
                            ignore_hosts]
        return hosts
18
    def _schedule(self, context, topic, spec_obj):
20
        elevated = context.elevated()
22      hosts = self.hosts_up(elevated, topic)
        if not hosts:
24          msg = _("Is the appropriate service running?")
            raise exception.NoValidHost(reason=msg)
26
        hosts = self._filter_hosts(hosts, spec_obj)
28      if not hosts:
            msg = _("Could not find another compute")
30          raise exception.NoValidHost(reason=msg)

32      return sorted(hosts)

34  def select_destinations(self, context, spec_obj):
            return self._schedule(context, CONF.compute_topic
                            , spec_obj)
```

3.2.2 Modification of Nova-API

When Nova receive a message of instance creation, the first method that is called is **create** (Line: 10) in the file: *nova/api/openstack/compute/servers.py*. We moved the code inside **create** in a new method **_create**, and in case the robustness is not in the body of the instances the **_create** method is used to build the instance with the default scheduler.

If the instance has the attribute "robustness", our **_Personalized_creation** (Line: 15) is called.

```
### servers.py
2

@wsgi.response(202)
4 @extensions.expected_errors((400, 403, 409))
@validation.schema(schema_server_create_v20, '2.0', '2.0')
6 @validation.schema(schema_server_create, '2.1', '2.18')
```

```
@validation.schema(schema_server_create_v219, '2.19', '2.31')
@validation.schema(schema_server_create_v232, '2.32', '2.36')
@validation.schema(schema_server_create_v237, '2.37')
def create(self, req, body):
    #CODE TO CHANGE THE SCHEDULER

    if"os:scheduler_hints" in body.keys():
        if 'robustness' in body['os:scheduler_hints']:
            return self._Personalized_creation(req, body)

    return self._create(req, body)
```

We focus just on the main part of the **_Personalized_creation** method (it has around 250 LOC); First of all we extract the value of the robustness from the body of the call (Line: 5);

Next step is to create the personalized scheduler(Line: 15), and get the list of the Compute IDs where to put the instances (Line: 16) In case the number of Hosts returned by the scheduler is not enough to satisfy the robustness, we rise an exception(Lines: 17-18).

If the Compute Nodes are enough, we create the instances in the Hosts returned by the scheduler (done inside the **for** cycle at Line: 21)

```
### servers.py

def _Personalized_creation(self, req, body):
    """Creates new instances using my scheduler."""
    robustness = int(body["os:scheduler_hints"]["robustness"]
                                )
    class personalized_Obj():
        def __init__(self, num_instances, ignore_hosts = []):
            self.num_instances = num_instances
            self.ignore_hosts = ignore_hosts

    spec_obj=personalized_Obj(body["server"]["max_count"])


    context = req.environ['nova.context']
    pers_schedule=RRScheduler()
    ll = pers_schedule.select_destinations(context, spec_obj)
                                [:robustness + 1]
    if len(ll) < robustness + 1:
```

```
        raise exc.HTTPBadRequest(explanation="Not enough Host
                                  for robustness = "+str(
                                 robustness))
    instances_created = []

    for i in ll:
        context = req.environ['nova.context']
        server_dict = body['server']
        ###CONTINUE WITH THE CREATION###
```

3.2.3 Keep track of the instances

Ones that we have multiple copies of the same instance, correctly distributed on the Compute nodes, we need to keep track of these instances.

Each instance has a unique ID, and by default, we don't know if an instance is a copy of a robust instance or a normal one.

So we created a new Database call "Robustness"; when a Robust instance is created, we create a RobustID and we assign it to each copy of the instance created.

The following code is executed after the allocation of the instances in the Compute Nodes, in the **_Personalized_creation** method in the file *nova/api/openstack/compute/servers.py*.

After the allocation, the IDs of the instances created, are stored in the list **instance_created**. We create an id for grouping all the copy of the same instance, and keep saving the robustness value **robustness_id** (Line: 3).

We save the tuple (robustness, robustness_ID) in the table robustness (Lines: 6-10), and the tuple (robustness_id, instance_id) in the table instances (Lines: 12-18).

```
### servers.py _Personalized_creation method

    robustness_id = str(uuid.uuid4())

    cur = db.cursor()
    try:
        cur.execute("""INSERT INTO robustness (robustness,
                                 robustness_ID) VALUES ( %s , %
                                 s );""" % ("'"+str(robustness)
                                 +"'", "'"+robustness_id+"'"))
```

```
8           db.commit()
    except:
10          cur.rollback()

12      for i in instances_created:
            try:
14              cur.execute("""INSERT INTO instances (
                                robustness_id, instance_id)
                                VALUES ( %s , %s );""" % (
                "'" + robustness_id + "', '" + str(i.__dict__['
                                obj']['server']['id']) + "'"))
16              db.commit()
            except:
18              cur.rollback()

20      chains = []
        with open("/var/log/nova/nova-chain.log", "a") as
                                chain_log:
22          chain_log.write('\n\n'+str(time.time())+" ADD chain
                                ID=%s, robustness %d" % (
                                chain_id, robustness))
```

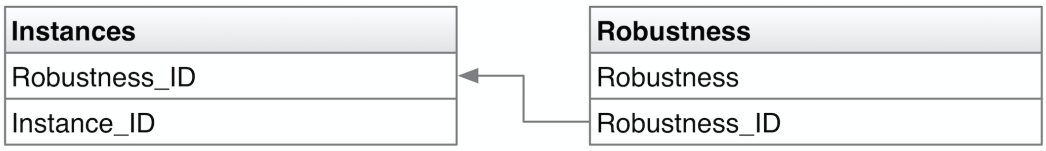

Figure 12: Database robustness

In case the user wants to remove the instance created (and the copies), it has to manually select all the copy of the instance and run the delete command from Horizon or the Nova-Client. We want automate this process, so we modify also the **delete** method in the file:
nova/api/openstack/compute/servers.py If we found the ID of the compute that the user wants to delete in the instances table, we select all the instance_id with the same robustness_id (Lines: 10-11), and delete all the instances(**for** cycle Line: 13)

```
### servers.py delete method

def delete(self, req, id):
        """Destroys a server."""

        # CODE BEFORE ...#
        #db robustness connected
        cur = db.cursor()
        chain_id = data[0][0]
        cur.execute("""SELECT * FROM instances WHERE chain_id
                                =%s""" % ("'"+robustness_id+"'
                                ",))
        instances = cur.fetchall()

        for i in instances:
            try:
                self._delete(req.environ['nova.context'], req
                                , i[1])
            except exception.InstanceNotFound:
                msg = _("Instance could not be found")
                raise exc.HTTPNotFound(explanation=msg)
            except exception.InstanceUnknownCell as e:
                raise exc.HTTPNotFound(explanation=e.
                                format_message())
           ### CONTINUE ...###
```

3.3 Test and Validation

First of all, we deployed OpenStack(Oca ta Version) in an infrastructure with one Controller node (in this case the Controller also manage the network), and multiple Compute nodes.

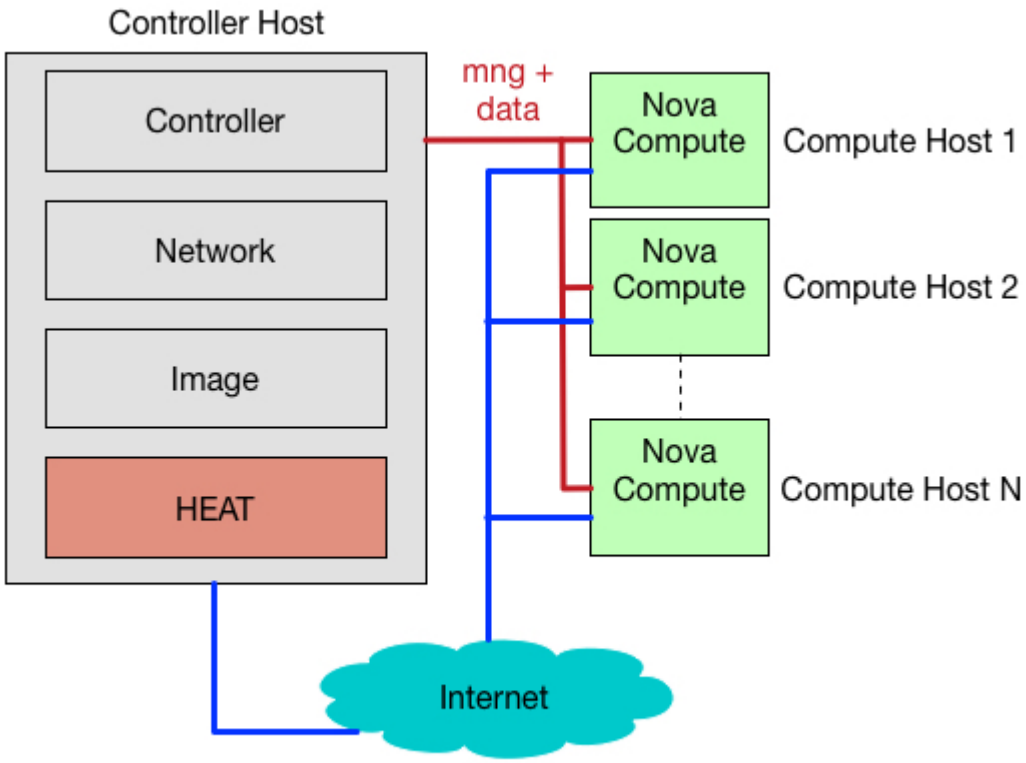

Figure 13: OpenStack Infrastructure

We tested the infrastructure to check the correctness of our installation[17], and after that, we installed HEAT(orchestration engine) to define chains (launch multiple composite instances based on templates in the form of text files). We can create a chain, but we cannot ensure the robustness yet. However, as we have seen in Section. 2.3, in order to guarantee the robustness, we need to replicate the functions of the chain and correctly choose in which Compute node build each network function.

We need to test our code before to insert the modification in our infrastructure; Fortunately, OpenStack provides a large number of unit tests(more than 16.000), each change in the code has to pass all the unit tests; If our code is compatible with all the unit tests, we modify the code inside our deployment.

To launch our code with the unit tests, we set up a virtual environment with the package "virtualenv"(a Python package that creates an environment, with all the packages needed that has its own installation directories, that doesn't share libraries with other virtualenv environments and optionally doesn't access the globally installed libraries either)[16][18].

Of course, we cannot launch all the unit test for each modification of the code, so to automate and standardize testing, we used Tox(a generic virtualenv management and test command line).

We can use Tox for:

- running tests in each of the environments, configuring your test tool of choice
- checking that package installs correctly with different Python versions and interpreters
- acting as a frontend to Continuous Integration servers, greatly reducing boilerplate and merging CI and shell-based testing.

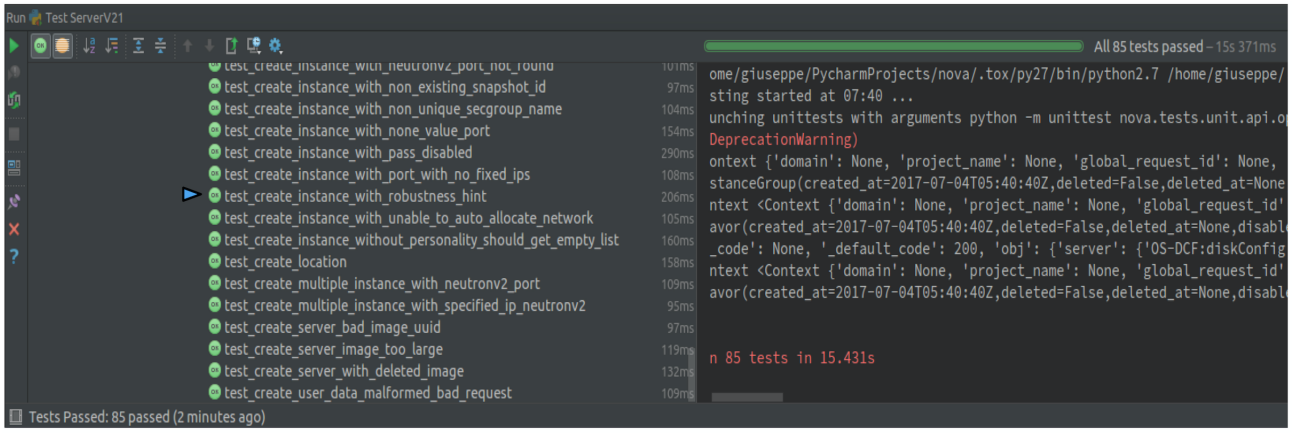

Figure 14: Subset of UnitTests for Nova-API

4 Demo

In this chapter we present a demo; In the infrastructure, we have one controller node and 3 compute nodes ("compute1","compute2","compute4"). The video of the demo is online at the link: OpenStack Demo.[19]

First, we test our infrastructure, with one normal instance.

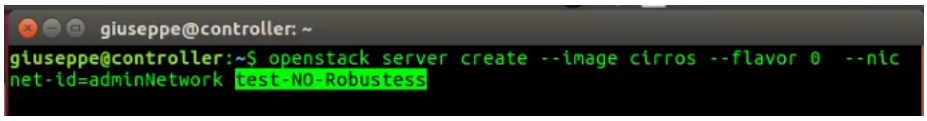

Figure 15: Creation of a norma instance with the CLI

We can see that the instance is correctly running from the Horizon web interface.

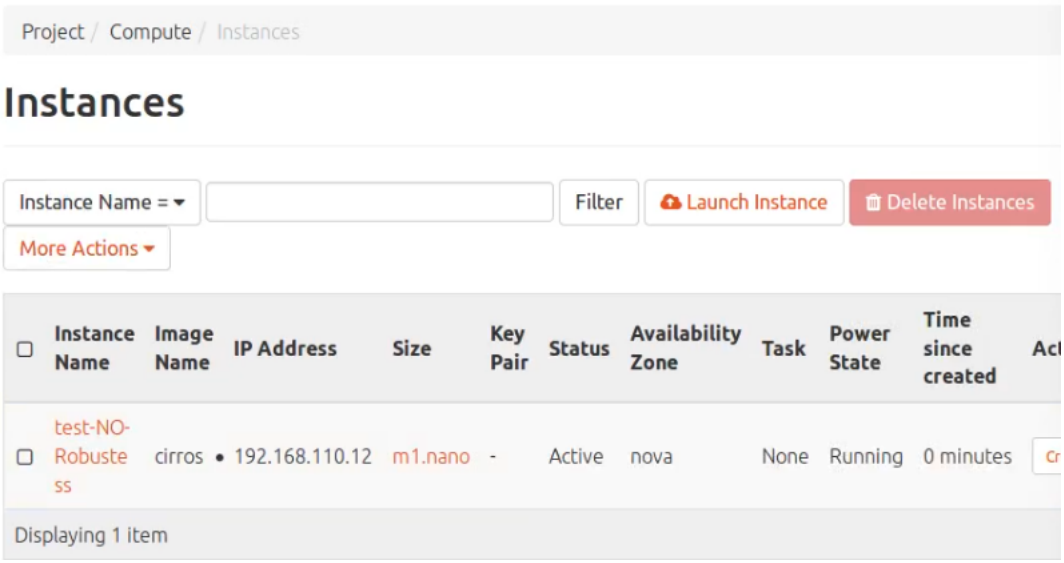

Figure 16: Check the interface created with Horizon GUI

Now we start a robust instance, with $robustness = 2$(so at the end we should have three instances created with the same properties).

```
giuseppe@controller:~$ openstack server create --image cirros --flavor 0  --nic
net-id=adminNetwork --hint robustness=2 test-Robustess
```

Figure 17: Creation of a robust instance

We can see from the web interface that 3 instances with the same name, network and flavor are created, the next step is to check if they are in the correct compute nodes.

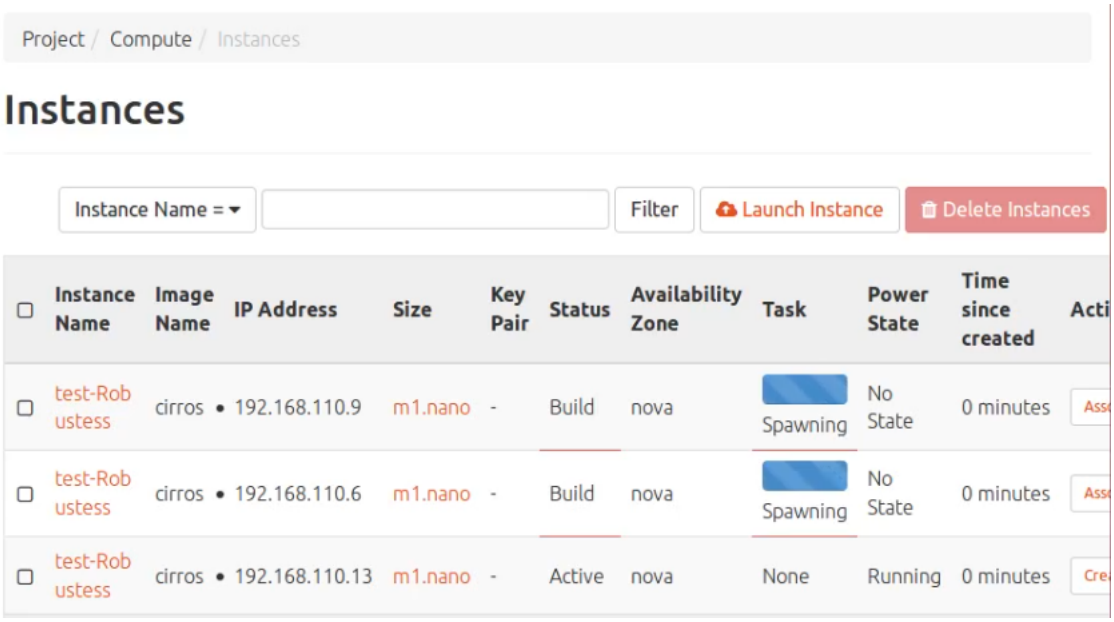

Figure 18: Check the creation of the robust instance with Horizon GUI

To check that the scheduler works correctly, we can see from the Horizon interface where the instances are running(only the admin can see where the instances are physically deployed).

We can see(Figure: 19) that the instances created are scheduled in different hosts(the scheduler is working correctly).

Figure 19: Check the placement in the admin Web interface

We checked that the Instances are correctly deployed, now we try to delete one instance from the web interface. If everything works fine, all the instances created with the robustness will be deleted automatically.

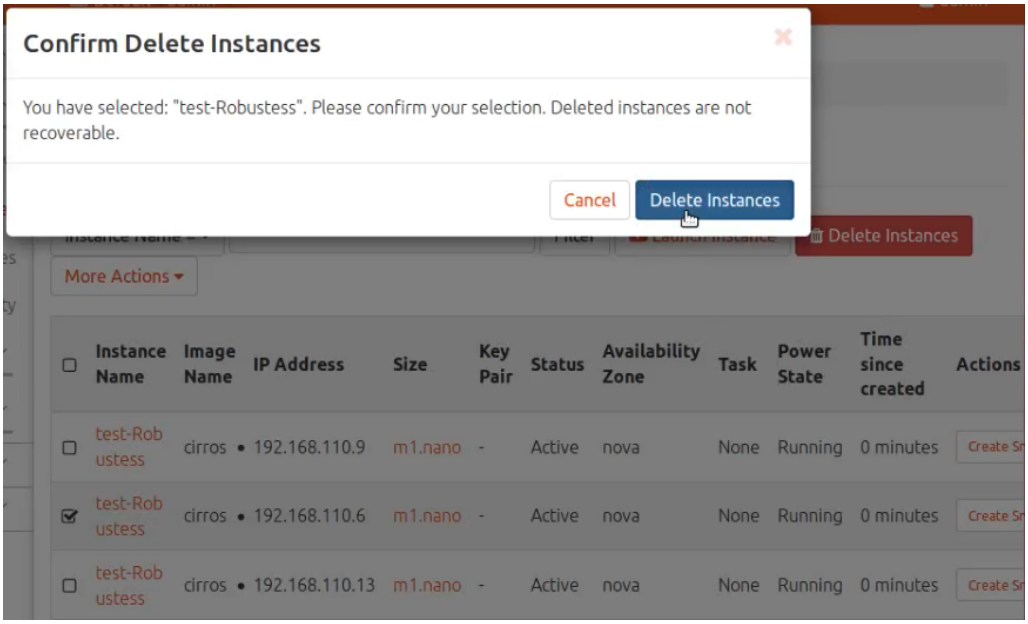

Figure 20: Delete one copy of the robust instance

We can see that all the instance created to guarantee the robustness are deleted.

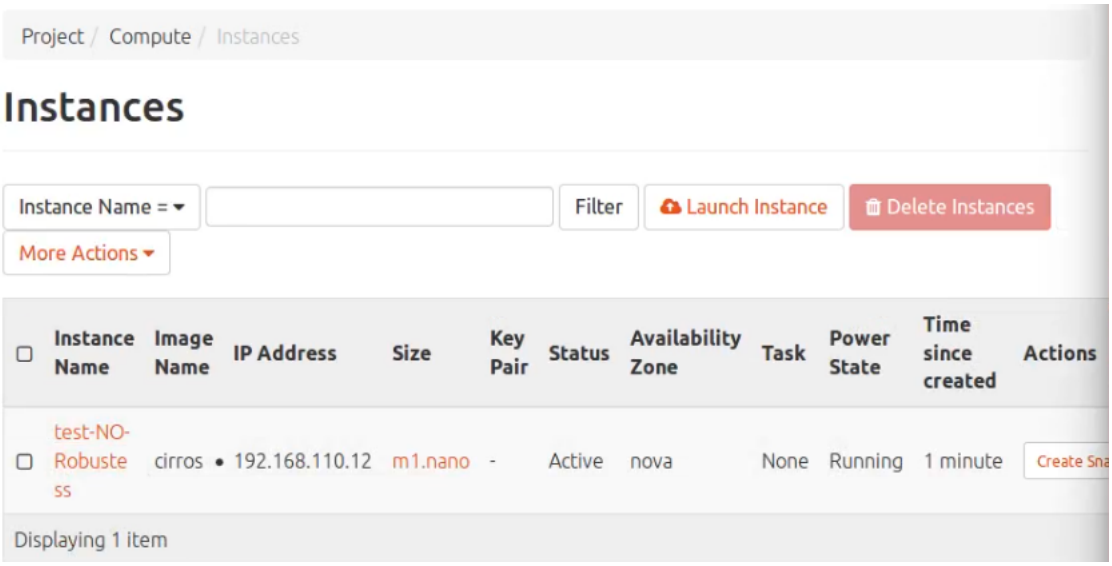

Figure 21: Check if all the copies are deleted

The last thing to check, if to see what happens if the robustness is too big for the infrastructure; Remember that in this example we have 3 compute nodes, so it is impossible to guarantee the $robustness >= 3$. Let's check what happens if we try to build an instance with $robustness = 3$

```
giuseppe@controller:~$ openstack server create --image cirros --flavor 0  --nic
net-id=adminNetwork --hint robustness=3 test-Robustess
Not enough Host for robustness = 3 (HTTP 400) (Request-ID: req-7edc4d36-8330-40f
1-b76f-5e34a53fbded)
giuseppe@controller:~$
```

Figure 22: Bild a instance with too much robustness

Correctly, we receive an error informing that the robustness is too big, and no instance is deployed.

5 Conclusion

OpenStack system provide an Infrastructure as a Service cloud with good scalability and manageability. In the current version (Ocata), the installation is manageable also for who is not used in system administration practice, the documentation for the final users is good and continuously updated by the OpenStack community.

For developers that want to start to contribute the situation is different. As we said, OpenStack is continuously evolving, and the documentation for the developers is not updated in parallel, with the OpenStack code and Architecture.

One solution to this problem can be to build an LTS(Long Term Version) version of OpenStack supported for a couple of years and in parallel developing new versions every six months (like the Ubuntu OS)

Regarding robust SNFC; we proposed different solutions, and we implemented the one that best fits for our goals; we can see that if a system is not designed with the robustness concept in mind since the beginning, it is difficult to add it later.

5.1 Future work

We created an environment that allows to multiply and schedule the instances of a SNFC, in the next step we need to create the connection between the instances created. Let's take following example; a user wants to create a chain of two functions $robustness = 1$.

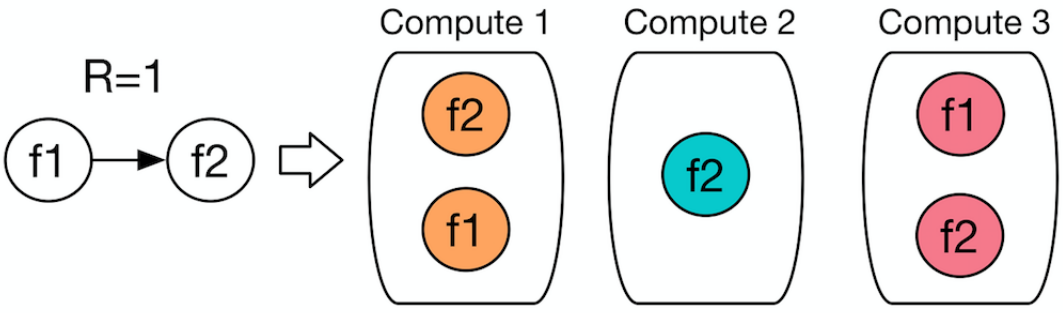

Figure 23: Example of placement

A scheduler implemented in our infrastructure decide to schedule and place the instances like in the previous figure.

At this point, the instances are correctly placed. They need to communicate each other; in this example, the input traffic should be directed to the instances f1, the output of the instances f1 directed to the input of the instances f2, and finally, the output of the instances f2 directed to the Output of the chain.

One idea can be to use Load Balancers between the Instances representing different functions; Fortunately, OpenStack (Neutron project), provide the API to build Load Balancers[20]. So the idea is to create load balancers after the creation of the instances, like in the following example.

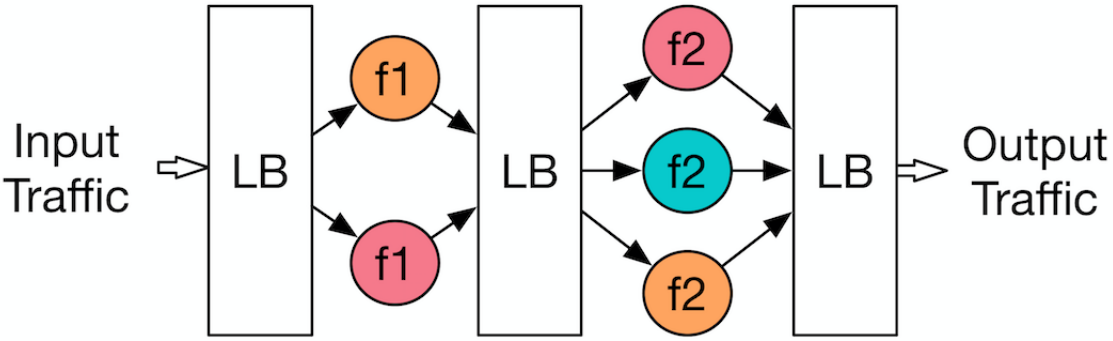

Figure 24: Load Balancers to split the traffic

References

[1] Peter Mell, Timothy Grance, *"The NIST Definition of Cloud Computing"*. Nacional Institute of Standards and Technology. September 2011.

[2] https://en.wikipedia.org/wiki/OpenStack, 28th August 2017

[3] https://www.openhub.net/p/openstack, 28th August 2017

[4] https://www.openstack.org, 28th August 2017

[5] Giuseppe Paternò, *"OpenStack - Nova e Glance"*. 14th December 2015, http://www.guruadvisor.net/it/cloud/132-openstack-nova-e-glance

[6] Alessandro Consalvi, *Studio e implementazione del paradigma SDN attraverso Openstack e Opencontrail.* 2016, University of L'Aquila.

[7] Andrea Tomassilli, *Virtual networking in OpenStack.* 2014, University of L'Aquila.

[8] https://docs.openstack.org/security-guide/networking/architecture.html, 28th August 2017

[9] Brad Topol, Henry Nash, Steve Martinelli, *"Identity, Authentication, and Access Management in OpenStack"*, December 2015.

[10] *"A quick introduction to OpenStack Heat"*, 31st January 2017. http://superuser.openstack.org/articles/quick-intro-openstack-heat/

[11] Ken Gray, Thomas D. Nadeau, *"Network Function Virtualization"*, 2016, Morgan Kaufmann.

[12] Rajendra Chayapathi, Syed F. Hassan, Paresh Shah, *"Network Functions Virtualization (NFV) with a Touch of SDN"*. 2016, Addison-Wesley Professional.

[13] Ghada Moualla, Thierry Turletti, Damien Saucez, *"To Split, or not to Split: When SFC Robustness is at Stake"*, 2017, Inria, France.

[14] docs.openstack.org/mitaka/config-reference/compute/scheduler

[15] Giuseppe Di Lena, https://github.com/Giuseppe1992/nova-deployed, Nova repository for the implementation of the Robustness.

[16] https://docs.openstack.org/infra/manual/developers.html

[17] Launch an instance, OpenStack Ocata Documentation

[18] Emily Hugenbruch, *"A guide to testing in OpenStack"*. 4th March 2016, developer.ibm.com

[19] Giuseppe Di Lena, *"Create a robust instance in OpenStack"* https://www.youtube.com/watch?v=8jtQhlSKPbM, Demo.

[20] https://wiki.openstack.org/wiki/Neutron/LBaaS